UFR de mathématique et d'informatique
Université de Strasbourg

MASTER'S DEGREE IN COMPUTER SCIENCE
SCIENCE AND ENGINEERING OF NETWORKS, INTERNET, AND SYSTEMS

Research Project (TER)

Lucian MOCAN
lucian.mocan@etu.unistra.fr

COMPILATION AND INTERPRETATION OF FUNCTIONS IN THE ALTHREAD LANGUAGE

May 6, 2025

Supervisor:

Quentin BRAMAS
bramas@unistra.fr

Contents

Chapter 1

Introduction

Althread is an educational programming language designed to model and verify distributed systems, such as applications operating across networked environments [1]. It addresses the limitations of traditional modeling tools, which, despite their robustness, often feature complex syntax and outdated designs that challenge novice learners. Althread introduces a C-inspired syntax, familiar to those with prior programming experience, while preserving the core capabilities of distributed system design: managing concurrency, facilitating inter-task communication, and addressing non-determinism [1].

This dissertation extends Althread by incorporating user-defined functions, enhancing its expressivity and applicability for educational purposes. These functions enable students to create reusable, modular code, simplifying the design and verification of complex systems. The work explores the significance of this contribution, detailing the challenges of distributed systems, the principles underpinning Althread's design, and the technical considerations of integrating user-defined functions into its framework.

The subsequent chapters begin with an overview of the state of the art in distributed systems and modeling tools, outlining current solutions and their limitations, particularly in educational settings. This is followed by a detailed discussion of Althread's architecture and design, focusing on its educational goals and the technical foundations of its compiler and virtual machine. Understanding these aspects is crucial as the implementation of user-defined functions requires modifications to the language's grammar and compilation process. The main contribution of this work, the implementation of user-defined functions, is then presented. This includes key design decisions, syntax and semantics, and integration with the existing language features. Finally, a series of examples and test cases demonstrate the functionality of the extension, and the work concludes with future directions for the continued development of Althread.

Chapter 2

State of the Art

2.1 Basics of Distributed Systems

Computers have evolved significantly: from single-CPU machines to powerful multi-core processors, from isolated devices to networks of connected systems. Their core tasks, memory management and arithmetic operations, haven't changed, yet consistent reliability remains essential. Once limited to a few scientists, technology now powers both research and daily life. Systems like self-driving cars, banking platforms, and streaming services depend on them, making their robustness critical.

To address these demands, distributed systems play a key role. Tanenbaum and van Steen define them as "a networked computer system in which processes and resources are sufficiently[1] spread across multiple computers" [2]. Well-designed distributed systems have several advantages [2]:

- **resource sharing**: Shared storage and computing power optimize costs and enable collaborative use.
- **transparency**: Users have a seamless experience without needing to understand the inner workings of the system (to a certain degree).
- **openness**: Integrate with other systems smoothly.
- **dependability**: Ensure availability, reliability, fault tolerance and security.
- **scalability**: Expand across multiple nodes with minimal performance degradation.

These benefits enable distributed systems to underpin modern computing. However, achieving them requires sophisticated algorithms to coordinate networked nodes, introducing significant complexity due to the decentralized nature of these systems. The following section examines these algorithms in detail.

2.2 Distributed Algorithms

Distributed algorithms orchestrate the interactions of networked computers, yet unlike a central conductor, each node operates independently while coordinating with others. Such algorithms have many applications. For example, Bitcoin's consensus algorithm [3] ensures agreement on transactions across nodes, while distributed locks in cloud storage systems like Google Cloud Storage manage data access [4].

When executed concurrently, these algorithms face complexity due to the lack of centralized control [5]. Key difficulties include:

- no **global state**: Nodes only know local data, complicating system-wide decisions.

[1] Here, "sufficiently" means to a degree that the system depends on multiple computers working together.

- no **global time-frame**: Asynchronous actions hinder synchronized execution.
- no **main coordinator**: Though it provides better fault tolerance, decentralized operation risks conflicting node behaviors.
- **non-determinism**: Unpredictable delays or failures disrupt coordination.

As Lynch notes, "Because of all this uncertainty, the behavior of distributed algorithms is often quite difficult to understand" [6]. The challenges of non-determinism, lack of global state, and asynchronous execution make it critical to formally model and verify these algorithms to ensure their correctness, as explored in the following section.

2.3 Models and Design Verification

Detecting flaws in the design of distributed systems early is crucial. Implementation-level testing often fails to uncover issues rooted in a system's logic or structure. A formal **model** is a mathematical representation of a system's behavior through states, events, or communication patterns [7, 8], and provides a foundation for verifying correctness prior to development.

Since the late 1970s, several formal methods and tools have been developed to build and verify these models.

CSP (Communicating Sequential Processes) [8] (1978), introduced by Hoare, models concurrent systems through synchronous communication between processes. CSP uses algebraic notation to describe how processes interact via channels, making it ideal for specifying communication protocols. Its influence is evident in modern languages like Go [9] and Erlang [10], and it supported the verification of International Space Station systems in 1999 [11]. However, CSP's abstract syntax can be challenging for beginners, requiring a strong grasp of process algebra.

SPIN [12] (1997), developed by Holzmann, is a model checker for asynchronous distributed systems. It uses **PROMELA** (Process Meta Language) to define system models, focusing on process interactions and communication behaviors [13]. SPIN exhaustively verifies properties like deadlock freedom or protocol correctness, making it valuable for applications like network protocol design. Yet, PROMELA's syntax, with constructs like guarded commands, can feel unintuitive to students familiar with procedural programming, limiting its accessibility in educational settings.

TLA+ [7] (1999), created by Lamport, specifies systems through state transitions and temporal properties, ensuring correctness across all possible executions. Its companion language, **PlusCal**, offers a more programmer-friendly syntax for describing concurrency and non-determinism, which is then translated into TLA+ for verification [7]. Amazon AWS has used TLA+ since 2011 to uncover critical bugs in its cloud infrastructure [14]. Despite its power, TLA+'s reliance on mathematical logic and temporal formulas poses a steep learning curve, particularly for those without formal methods training.

Despite their power, these tools present challenges, particularly for students or those new to formal methods. Their complex syntax, such as CSP's algebraic notation, PROMELA's guarded commands, and TLA+'s temporal logic, can be daunting and confusing. As distributed systems become increasingly vital, making these tools easier to learn is crucial. Althread addresses this challenge with its C-inspired syntax. The next section explores how Althread does this.

2.4 Althread's Architecture and Design

To extend Althread with user-defined functions, a clear understanding of its architecture is essential. As introduced earlier (Chapter 1), Althread is designed as an educational language for modeling distributed systems, focusing on intuitive syntax and the ability to simulate concurrent behavior through processes and channels. [1]. Its primary objectives include facilitating learning through a simple, C-inspired syntax, promoting accessibility with an open-source and cross-platform approach, enabling the modeling and verification of distributed systems, and supporting debugging through integrated tools [1].

The Althread compiler and its virtual machine are both implemented in Rust. This decision builds on previous development efforts and reflects a deliberate choice for a language offering strong guarantees of memory safety, high performance, and a growing community [15, 16].

This section presents the compiler and the virtual machine, which together form the foundation of Althread's aim to simplify distributed system design.

2.4.1 The Compiler

Syntax Overview

Before examining the grammar and the construction of the Abstract Syntax Tree (AST), it is helpful to first understand the surface syntax of Althread. Consider Figure 2.1, which shows an example program that highlights Althread's syntax and its use of channel-based communication:

```
shared { // block containing all global variables
    let A = 1;
    let B = 0;
    let Start = false; // synchronizes processes
}


program A() {    // program template A

    wait Start; // waits until Start == true

    // waits on the process' input channel
    wait receive in (x,y) => {
        print("received ", x, " ", y);
    };
}


main {

    // starts a process with program template A
    let pa = run A();
    let pb = run A();

    // creates and links an output channel to
    // the input channel of the process
    channel self.out (int, bool)> pa.in;
    channel self.out2 (int, bool)> pb.in;
```

```
    Start = true;
    send out (125, true); // send in the channel
    send out2 (125, false);
}
```

Figure 2.1: This Althread example demonstrates global variables, program templates, and channel-based communication. Two processes are spawned, each waiting for a global `Start` signal before receiving and printing a message (e.g., `received 125 true`) sent via channels.

Based on this surface syntax, the next step in the compilation process is to define a formal grammar that enables parsing and AST construction.

The Grammar

Althread's grammar is defined as a Parsing Expression Grammar (PEG). Compared to Context-Free Grammars (CFGs), which can produce ambiguous parse trees for complex syntax, PEGs ensure a single, unambiguous parse. In Althread, PEGs are processed using a Rust-based parser called `pest.rs` [17]. The following features of PEGs underpin Althread's grammar:

- **Ordered Choice**: In PEGs, the `|` operator represents an ordered choice, evaluating alternatives sequentially and selecting the first match [18]. This deterministic behavior eliminates backtracking (retrying parse alternatives), a common issue in CFGs where ambiguous rules require conflict resolution [19]. For example, in the following rule from Althread's grammar `main_block` is tried first:

  ```
  blocks = { main_block | global_block | condition_block | program_block }
  ```

- **No Ambiguity**: PEGs produce a single parse tree for any valid input, avoiding CFG ambiguities that can complicate semantic analysis [18, 19]. For instance, Althread's rule for binary expressions ensures a unique parse:

  ```
  binary_expr = { unary_expr ~ (binary_operator ~ unary_expr)* }
  ```

  This rule (where ~ denotes sequence) parses expressions like `x + y * z` unambiguously because `pest.rs` implements a Pratt parser [20], a top-down operator precedence parser effective for handling complex operator precedence and associativity.

- **Unlimited Lookahead**: PEGs support unlimited lookahead, allowing the parser to peek ahead without consuming tokens, which is ideal for complex constructs [18]. In Althread, this facilitates parsing nested structures, such as block comments:

  ```
  block_comment = { "/*" ~ (!"*/" ~ ANY)* ~ "*/" }
  ```

  This rule (where `!` denotes negation and `ANY` matches any character) uses lookahead to ensure the comment ends with `"*/"`.

Building the Abstract Syntax Tree (AST)

The parsing phase, implemented using `pest.rs` [17], generates parse nodes (rule-token pairs) that represent the syntactic structure of Althread code. The Abstract Syntax Tree (AST), a hierarchical representation of the code, is constructed by identifying top-level program blocks (`main`, `shared`, `program`, etc.) and recursively processing their internal components. This transforms the linear sequence of tokens into a tree structure that captures the relationships between code elements, preparing it for compilation. The full AST of the code in Figure 2.1 is available for reference in Appendix B.1.

Compilation Pipeline

The compilation process converts the AST into bytecode (intermediate instructions) for Althread's virtual machine. The compiler establishes a context to manage:

- The program's stack and scope information
- Global variable declarations and their visibility
- Channels (mechanisms for inter-process communication)
- Standard library bindings
- Information about the context: shared, atomic

Each AST node is processed according to its type, handling:

- Program blocks and local variables
- Channel declarations and communication operations
- Always[2]/never[3] conditions

The output is a compiled project containing bytecode, a global memory layout, and runtime verification conditions executable by the virtual machine [21].

2.4.2 The Virtual Machine

The Althread virtual machine (VM), a runtime environment for executing bytecode, uses a stack-based architecture with strong support for concurrency and channel-based communication [21, 22]. It manages program execution through several key mechanisms.

Each program maintains its own state, including a stack for local variables and an instruction pointer to track execution progress. The VM's concurrency model employs non-deterministic scheduling, where instructions are executed non-atomically by default by a single thread, interleaving instructions from multiple processes. Atomic operations and blocks ensure uninterrupted execution when needed. The model also supports synchronous channel communication for message passing between processes and uses runtime verification to enforce always/never conditions. The global state encompasses shared memory for global variables, channel states for inter-process communication, and the states of all running programs.

Althread's architecture supports predictable and concurrent execution, enabling user-defined functions to enhance its distributed systems modeling. The next section examines how state-of-the-art tools implement function-like constructs.

[2]always = check that a condition is met at each iteration.
[3]never = check that a condition is never met during the execution.

2.5 Function Implementation in Distributed System Modeling Tools

A recurring theme across tools like PROMELA, PlusCal, and CSP is their focus as design or algorithm languages, not traditional programming languages. When considering how to implement user-defined functions in Althread, these tools, alongside concurrent languages like Erlang and Go, offer interesting insights. Their approaches to function-like constructs motivate Althread's programmatic approach [2].

PROMELA, used by SPIN, foregoes traditional functions for *inline* constructs, macro-like abbreviations that define local parameter variables without isolating them in a new scope [23]. These offer efficiency, maintaining accurate line-number referencing over standard macros, as SPIN's documentation highlights [24]. A Fibonacci *inline* (Figure 2.2) demonstrates its approachable syntax. Alternatively, a `proctype` can act as a server, handling requests via global channels with user-provided local channels, though this adds overhead. Using process templates as functions is less viable due to significant costs [24]. SPIN's documentation clarifies: "The language targets verification of process interaction, not computational structures" [13].

PlusCal, paired with TLA+, employs procedures resembling Pascal-like structures, not traditional functions [25]. Its Fibonacci procedure (Figure 2.3) requires labels for control flow and a global variable for results, adding complexity. Looking at 2.2 it's easily noticeable that PROMELA looks somewhat accessible, whereas PlusCal's procedure syntax requires a lot of extra steps, like labeling for any control flow/ blocks and a separate global variable to return the result.

```
inline fibonacci(n, result)
{
    int i, a, b, temp;

    a = 0; b = 1; i = 2;

    if
    :: (n == 0) ->
            result = 0
    :: (n == 1) ->
            result = 1
    :: else ->
        do
        :: (i <= n) ->
            temp = a + b;
            a = b;
            b = temp;
            i++;
        :: else -> break;
        od;

        result = b;
    fi;
}
```

Figure 2.2: PROMELA Fibonacci inline macro

```
variables globalResult = 0;
procedure Fibonacci(n)
variable a = 0; b = 1; i = 1;
    result = 0;
begin
FibStart:
    if n = 0 then
        result := 0;
        goto FibDone;
    else
        FibWhile:
        while i < n do
            result := a + b;
            a := b;
            b := result;
            i := i + 1;
        end while;
        if n = 1 then
            result := 1;
        end if;
    end if;
FibDone:
    globalResult := result;
    return;
end procedure;
```

Figure 2.3: PlusCal Fibonacci algorithm

Finally, Erlang, a cornerstone of practical distributed systems, employs functional-style programming within its asynchronous concurrency model [26, 27]. Functions, defined with pattern matching and guards, provide a concise syntax for process behaviors. For example, a Fibonacci function in Erlang (Figure 2.4) leverages pattern matching for clarity and expressivity, contrasting with PROMELA's macro-like approach and PlusCal's labeled procedures. However, students must adapt to Erlang's functional paradigms, which can present a learning curve for beginners in educational settings.

```erlang
fib(N) when N >= 0 ->
    fib_iter(N, 0, 1).
fib_iter(0, A, _) -> A; % Base case: return F(0) or final result.
fib_iter(1, _, B) -> B; % Base case: return F(1).
fib_iter(N, A, B) ->
    % Compute next Fibonacci: F(n) = F(n-1) + F(n-2).
    fib_iter(N - 1, B, A + B).
```

Figure 2.4: Fibonacci function and pattern-matching in Erlang

2.6 Existing Function Implementation in Althread

Althread provides a built-in `print` function and methods on lists for adding, removing, and accessing elements, which are essential for basic operations and debugging. The `print` function leverages Rust's `print!` built-in, enabling clear and concise debugging output. Although Althread currently lacks an `assert` function, its documentation already describes its potential use [28].

These features are integral to Althread's functionality, providing essential tools for debugging and data manipulation. They ensure that Althread remains a robust and practical system for modeling distributed systems.

This defines the state of the art, paving the way for implementation details.

Chapter 3

User-Defined Functions in Althread

My development of user-defined functions for Althread was informed by a compilers course taken last semester, which established a theoretical foundation in compiler design. To gain practical insights into function implementation, I followed my advisor's recommendation to study Crafting Interpreters by Robert Nystrom [29]. This resource provided a structured, step-by-step approach to function implementation, enabling me to tackle the task incrementally. As Althread previously lacked functions, their addition required careful consideration of the language's educational and technical objectives.

This chapter will cover design considerations, implementation details, and challenges, along with examples demonstrating the functionality and impact of user-defined functions in Althread.

3.1 Key Design Questions

Before exploring the implementation details, we must address key design questions that shaped the development of user-defined functions in Althread. Given that PROMELA is the current language used in the University of Strasbourg's Distributed Algorithms course, it serves as the primary benchmark, naturally driving efforts to surpass its capabilities. However, this was not the sole factor, as the design was also informed by approaches to user-defined functions in other languages referenced in the State of the Art (Chapter 2).

3.1.1 The Need for User-Defined Functions in Althread

Both PROMELA and Althread are designed for modeling and verifying distributed systems. However, their design priorities differ based on their intended use. While PROMELA uses inline constructs for code reuse, Althread aims to implement fully-fledged user-defined functions to support its educational goals.

PROMELA's *inline* mechanism achieves code reuse through textual substitution. Unlike in C or C++, where inlines are used for optimization [30] and may or may not be substituted based on compiler decisions, PROMELA's inlines always take place and are substituted directly into the code [31, 32]. This approach works for PROMELA's verification-focused goals but has several limitations that would impact Althread's educational effectiveness. Specifically, *inline* constructs do not support recursion, lack local scoping (variables share the same context as the calling code), and have limited flexibility in returning computed values. These limitations make it difficult to express complex algorithms and can lead to potential conflicts and reduced clarity.

Althread's focus on education requires features that make distributed systems more accessible to students. User-defined functions would provide several benefits. They allow for better code organization through encapsulation, providing clear scope boundaries for variables. This reduces the risk of conflicts and improves code clarity. Additionally, user-defined functions support both iterative and recursive implementations, enabling the natural expression of algorithms commonly used in distributed systems. These benefits make user-defined functions a valuable addition to Althread, enhancing its role as both an educational tool and a practical system for modeling distributed systems.

3.1.2 Syntax and Semantics of User-Defined Functions

One of the key design questions is how to integrate user-defined functions into Althread's syntax and semantics. Several considerations were addressed, each with its own solution and rationale; however, the final chosen design is presented below.

Function Declaration and Definition

To maintain consistency with Althread's existing syntax and to ensure ease of learning, the syntax for function declaration and definition was designed to be intuitive and familiar to users. The following syntax was adopted:

```
// Syntax for a function with a return value
fn <function_name>
    (<param1>: <type1>, <param2>: <type2>) -> <return_type> {
    <statements>;
    return <expression>;
}

// Syntax for a function with no return value (void)
fn <function_name>(<param1>: <type1>, <param2>: <type2>) -> void {
    <statements>;
    // Optional: return;
}
```

Figure 3.1: This syntax includes the function name, a list of parameters with their types, the return type, and the function body enclosed in curly braces. This structure is similar to function declarations in languages like C and Rust, making it easier for students to understand and use. The `-> <return_type>` construct is particularly appealing because it aligns well with mathematical notation, clearly indicating that given a specific input, the function will produce a corresponding output.

This notation enhances readability and reinforces the conceptual understanding of functions as mappings from inputs to outputs. Key rules for function declaration and definition include:

- A function must be declared starting with the keyword `fn` followed by a function name, a list of arguments with `(identifier: datatype, ...)` or empty if no arguments, and a return type.
- The return type of a function shall be `void` or an existing datatype. For simplicity, a function can't have multiple return types (e.g. `-> int | float | bool` is not allowed).
- The return value's datatype should be the same as the function's declared return datatype.
- If the return datatype is `void`, then a `return_statement` is not required, but can be used as `return;` to exit the function early.

- A function must have a return value for all code paths.

Example:

```
fn sum(a: int, b: int) -> int {
  return a + b;
}

fn print_sum(a: int, b: int) -> void {
  print("Sum: " + (a + b));
}
```

Figure 3.2: The first function takes in a list of parameters of datatype `int` and returns an `int`, the result of the sum of the two passed parameters. The second function's return type is `void` and it prints the sum of the two passed parameters to the screen.

Function Calls and Execution

Function calls in Althread should follow a straightforward syntax:

```
result = function_name(arg1, arg2);
```

Arguments are passed by value, and the return value is assigned to a variable. The parser has to be extended to recognize function calls and generate the appropriate intermediate code. During execution, the virtual machine (VM) should handle the function call by pushing the current state onto the stack, executing the function, and then restoring the state. Return values are managed by storing them in a temporary variable and then assigning them to the caller's variable. Key rules for function calls and execution include:

- Function arguments are passed by value; that is, copies of the original values are provided to the function.
- Recursive calls are allowed inside functions.
- Multiple definitions of the same function name are not allowed.
- An indefinite amount of `return_statement` is allowed. Only the first one is going to be evaluated (similar to Python, C, C++).
- Calling a non-existent `function_name` throws an error.
- A function can only be called inside a valid `program`.

Error Handling and Debugging

Error handling for functions in Althread should provide clear and informative error messages. When an error occurs within a function, whether it is a syntax error, a compilation-time error, or a runtime error, the system must generate an error message that includes the line number, the line contents, and a description of the error. This detailed information should help users quickly identify and fix issues in their code.

Example:

```
fn example_function(a: int, b: int) -> int {
  if (a == 0) {
    return b;
  }
  return 2.5; // This line will cause an error due to type mismatch
}
```

Figure 3.3: In this example, the function `example_function` attempts to return a `float` value (2.5) when the return type is declared as `int`. The error handling mechanism in Althread should generate an error message indicating the line number where the error occurred, the contents of that line, and a description of the error (e.g., "Type mismatch: cannot return float when int is expected").

3.1.3 Integration with Existing Language Features

Another important design question is how to integrate user-defined functions with Althread's existing language features.

Compatibility with Concurrency

With Althread's focus on modeling and verifying distributed systems, it is essential to consider how user-defined functions should interact with its concurrency model. Functions are often categorized as pure or impure. A *pure* function produces the same output for the same inputs and has no side effects. An *impure* function may have side effects, such as modifying shared state (e.g., variables in the `shared` block) or interacting with the environment through operations like channel communication. While pure functions are typically easier to reason about and verify, impure functions are necessary to model distributed systems, where side effects enable communication and synchronization.

In Althread's concurrency model, instructions within functions should behave like those in a program's body, meaning they can interleave with instructions from other simulated processes. This interleaving occurs because Althread uses a single-threaded scheduler that randomly selects the next instruction from any running process. Functions are not atomic by default, though `atomic` blocks can ensure atomicity if needed. Impure functions, particularly those involving channel communication or shared state, are more affected by interleaving, as their side effects introduce non-determinism, whereas pure functions remain predictable regardless of interleaved execution.

For example, consider this pure function:

```
fn sum(a: int, b: int) -> int {
  return a + b;
}
```

Figure 3.4: This pure function always returns the sum of its inputs with no side effects.

In contrast, an impure function might involve concurrent operations:

```
fn increment_and_get() -> int {
    Counter = Counter + 1; // Counter is a global variable
    return Counter;
}
```

Figure 3.5: This impure function introduces non-determinism by modifying a global variable, which can cause race conditions due to instruction interleaving.

To maintain clarity and predictability, programmers should consider the trade-offs between pure and impure functions when designing their code in Althread. Pure functions are easier to verify and debug due to their lack of side effects, but impure functions are essential for modeling concurrent behaviors like message passing. Althread does not enforce restrictions on function behavior, allowing both pure and impure functions to coexist.

In conclusion, the integration of user-defined functions into Althread's concurrency model requires careful consideration of their pure or impure nature. By allowing both types of functions, Althread preserves flexibility for modeling distributed systems while relying on programmers to implement functions in a way that ensures predictable and debuggable behavior. This approach enhances Althread's role as both an educational tool and a practical system for modeling distributed systems.

Built-in Functions and Methods

As seen in 2.6, Althread provides a set of built-in functions and methods that are essential for basic operations and debugging. It is crucial to understand how these built-in functions and methods are implemented and to introduce the necessary modifications to integrate seamlessly with user-defined functions.

The focus of this work is on integrating user-defined functions into Althread's concurrency model. Extending the methods available for user-defined data types is an interesting feature but is beyond the scope of this project. While being able to add custom functions for a data type would enhance the language's flexibility, it introduces additional complexity and is not the primary goal of this work.

This section has addressed the key design questions that guided the development of user-defined functions in Althread. The following sections will delve into the implementation details, challenges, and examples that demonstrate the functionality and impact of these features.

3.2 Updating the Compiler

3.2.1 Grammar Changes

To support user-defined functions in Althread, several modifications were made to the grammar. Specifically, the list of blocks was extended to include a `function_block`, and a new rule was created for the function block to adhere to the chosen function syntax. Additionally, the statements list was completed with a `return_statement`. These changes were implemented in the `althread.pest` file.

The updated grammar rules are as follows:

```
blocks = { ... function_block }
function_block = { FN_KW ~ identifier ~ arg_list ~ RARROW ~ datatype ~ code_block }
return_statement = { RETURN_KW ~ expression? ~ ";" }
```

Figure 3.6: These changes leverage the existing rule for parameters (arg_list) and use the same body of instructions (code_block) as used for program templates. This ensures consistency and intuitive syntax for defining and using functions within Althread. For a complete reference, the full grammar is provided in the Appendix A.

3.2.2 Building the AST

To accommodate user-defined functions, the Abstract Syntax Tree (AST) in Althread needs to be extended. The pest.rs syntax parser returns pairs of rules, and by adding a new matching rule for a pair, it is possible to build the necessary parts that identify a function block in the AST.

First, the existing Ast data structure is extended with a new field representing the function blocks. This field is implemented as a hashmap that maps function names to their corresponding parameter list node, return datatype, and code block node. This structure allows for efficient lookup and management of function definitions.

When constructing the AST, we check if a function definition already exists. If it does, an error is returned: Function <function_name> is already defined. Otherwise, the function definition is added to the AST, and the AST display function is updated to verify successful construction.

For example, consider the max function shown in Figure 3.7, which returns the maximum value between two integers. The corresponding AST, shown in Figure 3.8, demonstrates how the function definition is accurately represented. Each syntactic element is correctly included in the AST, ensuring that the structure is complete and accurate. The full updates to the code are available in the appendix for your reference (see Appendix B.2).

This AST extension accurately represents user-defined functions, supporting subsequent compilation steps and integration with Althread.

```
fn max(a: int, b: int) -> int {
    if (a > b) {
      return a;
    } else {
      return b;
    }
}
```

Figure 3.7: A max function returning the max value between two ints.

```
max -> int
\-- if_control
    |-- condition
    |   \-- binary_expr
    |       |-- left
    |       |   \-- ident: a
    |       |-- op: >
    |       \-- right
    |           \-- ident: b
    |-- then
    |   |-- return
    |   |   \-- value:
    |   |       \-- ident: a
    \-- else
        |-- return
        |   \-- value:
        |       \-- ident: b
```

Figure 3.8: The built AST for the max function.

3.2.3 Compilation Pipeline

To compile user-defined functions, the existing compiler state needs to be extended. Specifically, the `CompilerState` data structure is updated to include a boolean `in_function`, which helps in defining local variables within a function body, and a hashmap to store function names and their corresponding definitions. A function definition is represented by the data structure shown in Figure 3.9.

```
pub struct FunctionDefinition {
    pub name: String,
    pub arguments: Vec<(Identifier, DataType)>,
    pub return_type: DataType,
    pub body: Vec<Instruction>,
    pub pos: Pos,
}
```

Figure 3.9: A function definition is the representation of a function stored in the compiler's state/context. It contains the name of the function, a vector of parameters and their corresponding datatypes, the function's return type, the compiled body (a vector of instructions), and the position of the first line of the function definition in the code text for debugging and error reporting reasons.

Function Definition

One of the main challenges encountered while implementing function definitions was correctly managing the program stack and the stack depth, which initially proved difficult. However, with time and practice, this aspect became more manageable. The implementation of this functionality takes place in the main `compile` function, located in `ast/mod.rs` (parts of the code can be found in Appendix B). Initially, the shared block is compiled, followed by the addition of the user-defined functions compilation implementation just before the programs are compiled. For each function block representing a function definition stored in the AST, the `in_function` parameter in the compiler state is set to `true`, and the stack depth is updated by 1 to properly represent a function call. Each argument is then pushed onto the program's stack.

Another challenge arose with the order of operations when compiling function definitions, particularly in the context of recursion. A check is performed to determine if the function already exists in the compiler's state. If it does not, the body of the function is compiled, and if no explicit return statement is found, a return instruction is added. After cleaning the stack and reverting to the previous depth, the function definition is then finalized and stored. Initially, this function definition was only created after the body was compiled. However, this approach failed for recursive functions because the definition was not available during the body's compilation. To resolve this, a preliminary version of the function definition without the compiled body is inserted into the compiler state before the body is compiled. This allows recursive calls within the body to be correctly recognized, after which the compiled body is added to complete the function definition.

Additionally, the compiler must be able to compile a return instruction. This also allows for verification of whether a return statement is inside a function's body or outside, based on the compiler's state previously set `in_function` field. If a return statement is found outside a function, an error is returned informing the user that a return statement cannot be outside a function. This is also where the return instruction is set up with an important field, has_value, which indicates whether this return is void or a value. The full implementation of the return statement can be viewed in the Appendix C.1.

Currently, this implementation lacks a check to ensure that all code paths require a return value. This would necessitate constructing a control flow graph and analyzing all possible paths to verify that each path ends with a return statement.

Function Call

To support user-defined function calls, the existing function call statement in fn_call.rs (parts of the code can be found in Appendix C.2) is modified. The compiler checks if the function name exists in the compiler's state. If it does not, an error is returned: undefined function <function_name>. Otherwise, the compiler verifies that the argument count matches the function signature and that the datatypes of the arguments are as expected. If everything checks out, the compiler adds a function call instruction to the vector of instructions and pushes a variable onto the stack for the return value of the function.

3.3 Extending the VM

After compilation, the entire code is passed as a data structure to the virtual machine (VM). This data structure, referred to as CompiledProject, includes a copy of the user-defined functions stored in the compiler state. The structure of CompiledProject is illustrated in Figure 3.10.

```
Ok(CompiledProject {
    global_memory,
    user_functions: state.user_functions.clone(),
    programs_code,
    always_conditions,
    stdlib: Rc::new(state.stdlib),
})
```

Figure 3.10: The CompiledProject data structure encapsulates the entire set of instructions compiled from the AST, including the function definitions with their compiled bodies. It also contains the global memory, which stores shared, global variables, and the always conditions, which are conditions that must always hold true and are used for invariant checking. Additionally, it includes Althread's standard library, which is utilized by existing methods on lists and other data structures. The user-defined functions are cloned from the compiler state, ensuring that the VM has access to all function definitions necessary for execution.

The VM starts by executing instructions sequentially, beginning with the main block. Each instruction is matched with its defined behavior in the next function in running_program.rs (The full contents of this file are available in Appendix D).The function call instruction has been extended to support user-defined function calls. The VM uses a simplified version of an activation record [33] represented by a call stack (named call_stack), a vector containing data structures of type StackFrame as shown in Figure 3.11.

When a function call instruction is matched, the following sequence occurs:

1. The VM first retrieves and validates the function's arguments from the stack as a tuple.

2. A new `StackFrame` is created and pushed onto the `call_stack`, storing:
   - The return instruction pointer (`return_ip`) pointing to the next instruction after the call.
   - The caller's frame pointer (`caller_fp`) for maintaining proper stack boundaries.
   - A reference to the caller's code segment (`caller_code`) to restore the execution context.
   - The expected return type (`expected_return_type`) for type checking the function's result.
3. The VM then sets up the new execution context by:
   - Setting the frame pointer (`frame_pointer`) to mark the current stack boundary.
   - Pushing the function arguments onto the stack above the new frame pointer.
   - Switching the current code segment to the function's compiled body.
   - Resetting the instruction pointer to 0.

```rust
struct StackFrame<'a> {
    return_ip: usize, // the instruction to return to
    caller_fp: usize, // the size of the stack
    caller_code: &'a [Instruction],
    expected_return_type: DataType
}
```

Figure 3.11: The `StackFrame` structure represents an activation record used to manage function calls. It includes the return instruction pointer (`return_ip`), the caller's frame pointer (the size of the stack before the function call, `caller_fp`), a reference to the caller's code segment (`caller_code`), and the expected return type (`expected_return_type`) for type checking the function's result.

When the function executes a return instruction, the process is reversed:

1. The return value is popped from the stack and type-checked against the expected return type. If there's a type mismatch, then the VM signals the error to the user and stops execution.
2. The stack is unwound to the caller's frame pointer.
3. The execution context is restored using the saved `StackFrame`.
4. The return value is pushed onto the caller's stack.

3.4 Testing

To validate the implementation of user-defined functions in Althread, I conducted a series of tests. These tests focused on ensuring that the compiler and VM correctly handle function definitions, calls, and returns, particularly in scenarios involving recursion, iteration, concurrency, and shared variable access.

3.4.1 Recursive and Loop-Based Functions

I chose the Fibonacci sequence as a test case because it effectively demonstrates the handling of recursion, iteration, and local variable definitions. Two versions of the Fibonacci function were implemented: a recursive version and an iterative version. These implementations ensure that the VM can correctly manage stack frames for recursive calls and efficiently handle loops and local variables.

The recursive Fibonacci function computes the Fibonacci number for a given input by calling itself with a decremented argument until the base case is reached. The iterative Fibonacci function uses a loop to compute the Fibonacci number, demonstrating the handling of iterative constructs and local variable definitions. The full code for both implementations can be viewed in Appendix E.1.

The main program calls both the recursive and iterative Fibonacci functions and prints the results. The expected output is:

```
Fibonacci recursive of 10: 55
Fibonacci iterative of 10: 55
```

This output confirms that both the recursive and iterative implementations of the Fibonacci sequence work correctly in Althread. The VM successfully handled the recursive calls, demonstrating the effectiveness of the call stack and frame mechanism. Additionally, the iterative implementation confirmed that the VM can efficiently handle loops and local variable definitions.

3.4.2 Concurrent Message Processing

Another key test involved implementing a concurrent message processing scenario to demonstrate the use of atomic blocks, conditional statements, and access to shared variables. In this test, two worker processes are spawned, each waiting to receive a message. Upon receiving a message, each worker updates shared variables within an atomic block to ensure that updates are performed without interference from other processes. The full code for this implementation can be viewed in Appendix E.2.

The expected output is:

```
Processing message: value=125, flag=true
Processing message: value=125, flag=false
Channel test successful!
```

or

```
Processing message: value=125, flag=false
Processing message: value=125, flag=true
Channel test successful!
```

This output confirms that the concurrent message processing scenario works correctly in Althread, demonstrating the effective handling of atomic blocks, conditional statements, and shared variable access.

Chapter 4

Conclusion

This research focused on extending Althread, an educational programming language for distributed systems developed at the University of Strasbourg, by implementing user-defined functions. The primary goal was to enhance the language's capabilities through improved modularity and code reusability, making it more accessible and practical for students learning distributed systems programming.

4.1 Summary of Contributions

The project successfully integrated user-defined functions into Althread's existing architecture, marking a significant enhancement to the language's functionality. The implementation required comprehensive modifications to both the compiler and virtual machine (VM) to support function definitions, calls, and returns. Notable features include support for recursive functions, iterative constructs, and concurrent execution capabilities. To maintain the language's educational value, the error reporting system was expanded to provide clear, contextual feedback for function-related issues during syntax checking, compilation, and execution phases. The implementation was validated through various test cases, including recursive and iterative Fibonacci implementations and concurrent message-processing scenarios. While these tests demonstrated the reliability of core functionalities, further testing of complex patterns and edge cases remains necessary.

4.2 Future Perspectives

The implementation of user-defined functions opens several promising avenues for future development. A priority enhancement would be the implementation of control flow graph analysis to ensure complete return value coverage across all code paths. This would strengthen the language's reliability and help prevent runtime errors.

The current implementation could be further enhanced by adding support for function modularity across files. This would require modifying Althread's grammar and extending the compiler to read and compile external functions into its state. Additionally, an interesting idea would be incorporating advanced programming features such as lambda functions and pattern matching, similar to Erlang's implementation. However, these additions would require careful evaluation of their compatibility with Althread's current grammar and pedagogical objectives.

An existing error in Althread's grammar regarding the parsing of comparison expressions (such as `v >= 1`) needs attention. While workarounds exist, such as using equivalent expressions like `v > 1 || v == 1` or `v > 0.99`, a proper solution would involve restructuring the grammar's expression handling system and the AST building process.

These proposed enhancements would strengthen Althread's position as both an educational tool and a practical platform for distributed systems modeling, while maintaining its accessibility for students and encouraging community involvement in its continued development.

Chapter 5

Bibliography

[1] Althread. *Introduction to Althread Guide*. 2025. URL: https://althread.github.io/en/docs/guide/intro/.

[2] Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems 4th edition*. 2025. URL: https://www.distributed-systems.net/index.php/books/ds4/.

[3] Ling Ren. *Analysis of Nakamoto Consensus*. Cryptology (ePrint) Archive, Paper 2019/943. 2019. URL: https://eprint.iacr.org/2019/943.

[4] Ahmet Alp Balkan. *Implementing Leader Election with Google Cloud Storage*. 2021. URL: https://cloud.google.com/blog/topics/developers-practitioners/implementing-leader-election-google-cloud-storage.

[5] Gerard Tel. *Introduction to Distributed Algorithms*. 2nd ed. Cambridge University Press, 2000.

[6] Nancy A. Lynch. *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers, 1996. ISBN: 978-1-55860-348-6.

[7] Leslie Lamport. *A High-Level View of TLA+*. https://lamport.azurewebsites.net/tla/high-level-view.html.

[8] C. A. R. Hoare. "Communicating sequential processes". In: *Commun. ACM* 21.8 (Aug. 1978), pp. 666–677. ISSN: 0001-0782. DOI: 10.1145/359576.359585. URL: https://doi.org/10.1145/359576.359585.

[9] Rob Pike. *Go Concurrency Patterns*. 2012. URL: https://go.dev/talks/2012/concurrency.slide#9.

[10] Joe Armstrong. "Erlang". In: *Communications of the ACM* 53.9 (2010), pp. 68–75. DOI: 10.1145/1810891.1810910. URL: https://cacm.acm.org/research/erlang/.

[11] Jan Peleska and Bettina Buth. "Formal Methods for the International Space Station ISS". In: *Correct System Design: Recent Insights and Advances*. Ed. by Ernst-Rüdiger Olderog and Bernhard Steffen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 363–389. ISBN: 978-3-540-48092-1. DOI: 10.1007/3-540-48092-7_16. URL: https://doi.org/10.1007/3-540-48092-7_16.

[12] Gerard J. Holzmann. "The model checker SPIN". In: *IEEE Transactions on Software Engineering* 23.5 (1997). URL: https://spinroot.com/spin/Doc/ieee97.pdf.

[13] Gerard J. Holzmann. *SPIN Model Checker Manual*. 2017. URL: https://spinroot.com/spin/Man/Manual.html.

[14] Chris Newcombe et al. *Formal Methods at Amazon Web Services*. Tech. rep. Amazon Web Services, 2015. URL: https://lamport.azurewebsites.net/tla/formal-methods-amazon.pdf.

[15] GitHub. *The top programming languages - The 2022 GitHub Octoverse*. 2022. URL: https://octoverse.github.com/2022/top-programming-languages.

[16] Stack Overflow. *Stack Overflow Developer Survey 2023*. June 2023. URL: https://survey.stackoverflow.co/2023/#section-admired-and-desired-programming-scripting-and-markup-languages.

[17] pest Project Contributors. *pest. The Elegant Parser*. pest v2.8.0. URL: https://docs.rs/pest/latest/pest/.

[18] Bryan Ford. "Parsing Expression Grammars: A Recognition-Based Syntactic Foundation". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Venice, Italy: ACM, 2004. DOI: 10.1145/964001.964011. URL: https://bford.info/pub/lang/peg.pdf.

[19] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools*. 2nd ed. Section 4.2.5, Context-Free Grammar: Ambiguity. Pearson Education, 2006.

[20] Vaughan R. Pratt. "Top Down Operator Precedence". In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '73)*. ACM, 1973, pp. 41–51. DOI: 10.1145/512927.512932. URL: https://tdop.github.io/.

[21] Althread Project Contributors. *Internal Architecture Guide*. URL: https://althread.github.io/docs/guide/internal/architecture.

[22] Althread Project Contributors. *Internal Guide: Virtual Machine*. URL: https://althread.github.io/docs/guide/internal/vm.

[23] Bernhard Beckert. *Formal Specification and Verification: Introduction to Promela*. Tech. rep. Based on a lecture by Wolfgang Ahrendt and Reiner Hähnle at Chalmers University, Göteborg. Karlsruhe Institute of Technology, 2009. URL: https://formal.kastel.kit.edu/beckert/teaching/Formale-Verifikation-SS09/11Promela.pdf.

[24] Gerard J. Holzmann. *Procedures*. Online. 2004. URL: https://spinroot.com/spin/Man/procedures.html.

[25] Leslie Lamport. *A PlusCal User's Manual: C-Syntax Version 1.8*. Available online at: https://lamport.azurewebsites.net/tla/c-manual.pdf. TLA+ Community, Mar. 2024.

[26] Ericsson AB. *Erlang System Documentation v27.3.3 - Concurrency in Erlang*. 2025. URL: https://www.erlang.org/doc/system/conc_prog.html.

[27] Ericsson AB. *Erlang Run-Time System (ERTS) v15.2.6 - Erlang Communication*. 2025. URL: https://www.erlang.org/doc/apps/erts/communication.html.

[28] Althread Project. *Althread Guide: Testing*. 2025. URL: https://althread.github.io/docs/guide/test.

[29] Robert Nystrom. *Crafting Interpreters*. 2021. URL: https://craftinginterpreters.com/.

[30] cppreference.com contributors. *Inline specifier*. 2024. URL: https://en.cppreference.com/w/cpp/language/inline.

[31] Computer Science and Michigan State University Engineering. *Inline*. 1997. URL: https://www.cse.msu.edu/~cse470/PromelaManual/inline.html.

[32] nimble-code. *Spin: A modeling and verification tool - source code (GitHub repository)*. URL: https://github.com/nimble-code/Spin/blob/master/Src/spinlex.c.

[33] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools*. 2nd ed. Section 7.2.2, Activation Records. Pearson Education, 2006.

Appendix A

Updated Grammar

The full grammar for Althread, including the modifications for user-defined functions, is provided below. This grammar is implemented in the `althread.pest` file.

```
/// filepath: althread.pest

/// # Althread Grammar
/// This file defines the grammar for the Althread programming language,

program = _{ SOI ~ blocks* ~ EOI }

/// ## Program Structure
/// The main building blocks of an Althread program are:
/// - **Main Block**: The entry point of the program.
/// - **Shared Block**: Declares global variables accessible across different parts of the program.
/// - **Condition Block**: Monitors conditions at each atomic step (e.g., always, never, eventually).
/// - **Program Block**: Encapsulates code that runs concurrently in parallel processes.
/// - **Function Block**: User-defined functions
blocks = _{ main_block | global_block | condition_block | program_block | function_block }

global_block    = { GLOBAL_KW ~ code_block }
condition_block = { condition_keywords ~ expression_block }
program_block   = { PROGRAM_KW ~ identifier ~ arg_list ~ code_block }
main_block      = { MAIN_KW ~ code_block }
// Functions
function_block = { FN_KW ~ identifier ~ arg_list ~ RARROW ~ datatype ~ code_block }

code_block           = { "{" ~ statement* ~ "}" }
expression_block     = { "{" ~ expression_statement* ~ "}" }
expression_statement = { expression ~ ";" }

condition_keywords = _{ ALWAYS_KW | NEVER_KW }

/// ## Statements
/// Statements are the executable instructions in the language.
/// They include assignments, declarations, expressions, print statements,
/// function calls, and control flow structures.

statement = {
    assignment_statement
  | declaration_statement
  | channel_declaration_statement
```

```
    | run_statement
    | send_statement
    | wait_statement
    | atomic_statement
    | if_control
    | for_control
    | loop_control
    | while_control
    | call_statement
    | code_block
    | break_loop_statement
    | return_statement
}


break_loop_statement = { (BREAK_KW | CONTINUE_KW) ~ identifier? ~ ";" }
assignment_statement  = _{ assignment ~ ";" }
declaration_statement = _{ declaration ~ ";" }
wait_statement        =  { WAIT_KW ~ (
      waiting_block
    | waiting_block_case) }
atomic_statement        =  { (ATOMIC_KW | "!") ~ statement }
call_statement        = _{ fn_call ~ ";" }
run_statement         = _{ run_call ~ ";" }
send_statement         = _{ send_call ~ ";" }
channel_declaration_statement = _{ channel_declaration ~ ";" }
// Functions
return_statement = { RETURN_KW ~ expression? ~ ";" }

fn_call  = { object_identifier ~ tuple_expression }
run_call = { RUN_KW ~ identifier ~ tuple_expression }
send_call = { SEND_KW ~ object_identifier ~ tuple_expression }
channel_declaration = {
    CHANNEL_KW ~
    object_identifier ~
    "<"? ~
    type_list ~
    ">"? ~
    object_identifier }

type_list = { "(" ~ datatype ~ ("," ~ datatype)* ~ ")" }
pattern_list = { "(" ~ pattern ~ ("," ~ pattern)* ~ ")" }
arg_list = {
  ( "(" ~ ")")
  | ("(" ~ (identifier ~ ":" ~ datatype) ~ ("," ~ identifier ~ ":" ~ datatype)* ~ ")" )
}
pattern = { identifier | literal }

/// ### Assignments
/// Assignments assign values to variables.
/// - **Unary Assignments**: Increment or decrement a variable (e.g., a++).
/// - **Binary Assignments**: Assign the result of an expression to a variable (e.g., a = b + c).
assignment = { binary_assignment }
```

```

side_effect_expression    = { run_call | fn_call | expression | ("[" ~ range_expression ~ "]") }

binary_assignment          = {
    identifier ~
    binary_assignment_operator ~
    side_effect_expression }
binary_assignment_operator = { ASSIGN_OP | ADD_ASSIGN_OP | SUB_ASSIGN_OP | MUL_ASSIGN_OP | DIV_ASSIGN_OP | MOD_ASSIGN_OP }

/// ### Declarations
/// Declarations introduce new variables, which can be mutable (let) or immutable (const).
declaration         = { declaration_keyword ~ identifier ~ (":" ~ datatype)? ~ ("=" ~ side_effect_expression)? }
declaration_keyword = { LET_KW | CONST_KW }

receive_expression = { RECEIVE_KW ~ object_identifier? ~ pattern_list ~ ("=>" ~ statement)? }

/// ### Expressions
/// Expressions evaluate values based on arithmetic and logical operations, following standard precedence rules.
expression = {
    fn_call
  | binary_expression
  | unary_expression
  | primary_expression
}

tuple_expression = {
 ("(" ~ ")") | ("(" ~ expression ~ ("," ~ expression)* ~ ")")
}
range_expression = {
  (expression ~ LIST_OP ~ expression)
}

primary_expression = _{ literal | identifier | "(" ~ expression ~ ")" }

unary_expression = _{ unary_operator? ~ primary_expression }
unary_operator   =  { POS_OP | NEG_OP | NOT_OP }

binary_expression   = _{ unary_expression ~ (binary_operator ~ unary_expression)* }
binary_operator     = _{ or_operator | and_operator | equality_operator | comparison_operator | term_operator | factor_operator }
or_operator         =  { OR_OP }
and_operator        =  { AND_OP }
equality_operator   =  { EQ_OP | NE_OP }
comparison_operator =  { LT_OP | GT_OP | LE_OP | GE_OP }
term_operator       =  { ADD_OP | SUB_OP }
factor_operator     =  { MUL_OP | DIV_OP | MOD_OP }

waiting_block = {
```

```
    (SEQ_KW | FIRST_KW) ~ "{" ~ waiting_block_case* ~ "}"
}
waiting_block_case       = {
    (receive_expression | expression)
    ~ (";" | ("=>" ~ statement)) }

/// ### Control Flow
/// Control flow structures include conditional execution and loops.

if_control    = { IF_KW ~ expression ~ code_block ~ (ELSE_KW ~ (if_control | code_block))? }
while_control = { WHILE_KW ~ expression ~ code_block }
loop_control = { LOOP_KW ~ statement }
for_control = { FOR_KW ~ identifier ~ "in" ~ list_expression ~ statement }

list_expression = _{ (range_expression | expression) }
/// ## Tokens
/// This section defines the keywords, operators, datatypes, and other tokens used in Althread.

/// ### Keywords
/// Keywords define the core constructs of the language.
KEYWORDS = _{
    MAIN_KW
  | GLOBAL_KW
  | PROGRAM_KW
  | ALWAYS_KW
  | NEVER_KW
  | RUN_KW
  | LET_KW
  | CONST_KW
  | IF_KW
  | ELSE_KW
  | WHILE_KW
  | FN_KW
  | RETURN_KW
  | BOOL
  | INT_TYPE
  | FLOAT_TYPE
  | STR_TYPE
  | VOID_TYPE
}

MAIN_KW    = _{ "main" }
GLOBAL_KW  = _{ "shared" }
PROGRAM_KW = _{ "program" }
WAIT_KW    = _{ "wait" }
ALWAYS_KW  =  { "always" }
NEVER_KW   =  { "never" }
RUN_KW     = _{ "run" }

FIRST_KW = { "first" }
SEQ_KW   = { "seq" }

LET_KW   = { "let" }
CONST_KW = { "const" }
```

```

IF_KW     = _{ "if" }
ELSE_KW   = _{ "else" }
WHILE_KW  = _{ "while" }
FOR_KW = _{ "for" }
LOOP_KW = _{ "loop" }
BREAK_KW = { "break" }
CONTINUE_KW = { "continue" }

SEND_KW = _{ "send" }
RECEIVE_KW = _{ "receive" }
CHANNEL_KW = _{ "channel" }

TRUE_KW   = _{ "true" }
FALSE_KW  = _{ "false" }
NULL_KW   = _{ "null" }

ATOMIC_KW = _{ "atomic" }

// Functions
FN_KW = _{ "fn" }
RETURN_KW = _{ "return" }
RARROW = { "->" }

/// ### Operators
/// Operators are used for arithmetic, logical operations, and assignments.
POS_OP = { "+" }
NEG_OP = { "-" }
NOT_OP = { "!" }

ADD_OP = { "+" }
SUB_OP = { "-" }
MUL_OP = { "*" }
DIV_OP = { "/" }
MOD_OP = { "%" }

EQ_OP  = { "==" }
NE_OP  = { "!=" }
LT_OP  = { "<" }
GT_OP  = { ">" }
LE_OP  = { "<=" }
GE_OP  = { ">=" }
AND_OP = { "&&" }
OR_OP  = { "||" }

LIST_OP = _{ ".." }

ASSIGN_OP     = { "=" }
ADD_ASSIGN_OP = { "+=" }
SUB_ASSIGN_OP = { "-=" }
MUL_ASSIGN_OP = { "*=" }
DIV_ASSIGN_OP = { "/=" }
MOD_ASSIGN_OP = { "%=" }
OR_ASSIGN_OP  = { "|=" }

/// ### Datatypes
```

```
/// Datatypes supported in Althread include boolean, integer, float,
    string, and void.
datatype    = { BOOL_TYPE | INT_TYPE | FLOAT_TYPE | STR_TYPE |
    VOID_TYPE | LIST_TYPE | PROCESS_TYPE }
BOOL_TYPE   = { "bool" }
INT_TYPE    = { "int" }
FLOAT_TYPE = { "float" }
STR_TYPE    = { "string" }
VOID_TYPE   = { "void" }
PROCESS_TYPE = { "proc" ~ "(" ~ identifier ~ ")" }
LIST_TYPE   = { "list" ~ "(" ~ datatype ~ ")" }

/// ### Literals
/// Include literals such as booleans, integers, floats, strings, and
     null.
literal =  { BOOL | FLOAT | INT | STR | NULL }
BOOL     = @{ TRUE_KW | FALSE_KW }
INT      = @{ ASCII_DIGIT+ }
FLOAT    = @{ ASCII_DIGIT+ ~ "." ~ ASCII_DIGIT+ }
STR      = @{ "\"" ~ (!"\"" ~ ANY)* ~ "\"" }
NULL     = @{ NULL_KW }

/// ### Identifiers
/// Identifiers are used for naming variables, functions, and other
    constructs.
identifier        = { IDENT }
object_identifier = { (IDENT ~ "." ~ object_identifier) | IDENT }

reserved_keywords = { (KEYWORDS | datatype) ~ !IDENT_CHAR }

IDENT      = @{ !reserved_keywords ~ ASCII_ALPHA ~ IDENT_CHAR* }
IDENT_CHAR = _{ ASCII_ALPHANUMERIC | "_" }

/// ## Whitespace and Comments
/// Whitespace and comments are ignored by the parser.
WHITESPACE = _{ " " | "\t" | NEWLINE }
NEWLINE    = _{ "\n" | "\r" | "\r\n" }

COMMENT        = _{ INLINE_COMMENT | BLOCK_COMMENT }
INLINE_COMMENT = _{ "//" ~ (!NEWLINE ~ ANY)* }
BLOCK_COMMENT  = _{ "/*" ~ (!"*/" ~ ANY)* ~ "*/" }
```

Figure A.1: Althread's full grammar updated to support user-defined functions

Appendix B

Updated AST Code

The full AST for the code in Figure 2.1:

```
shared
|-- decl
|    |-- keyword: let
|    |-- ident: A
|    \-- value
|         \-- int: 1
|-- decl
|    |-- keyword: let
|    |-- ident: B
|    \-- value
|         \-- int: 0
\-- decl
     |-- keyword: let
     |-- ident: Start
     \-- value
          \-- bool: false

main
|-- decl
|    |-- keyword: let
|    |-- ident: pa
|    \-- value
|         \-- run: A
|-- decl
|    |-- keyword: let
|    |-- ident: pb
|    \-- value
|         \-- run: A
|-- channel decl
|-- channel decl
|-- binary_assign
|    |-- ident: Start
|    |-- op: =
|    \-- value:
|         \-- bool: true
|-- send
|    |-- out
|    \-- tuple
|         \-- int: 125
|         \-- bool: true
\-- send
     |-- out2
     \-- tuple
          \-- int: 125
          \-- bool: false

A
|-- wait_control
```

```
|   \-- wait case
|       |-- ident: Start
\-- wait_control
    \-- wait case
        |-- receive
        |   |--  channel 'in'
        |   |--  patterns (x,y)
        |   |-- print
        |   |   \-- tuple
        |   |       \-- string: "received "
        |   |       \-- ident: x
        |   |       \-- string: " "
        |   |       \-- ident: y
```

Figure B.1: The AST structure can be viewed through a AST display implementation for all the nodes of the AST.

The full AST for Althread, including the modifications for user-defined functions, is provided below. This build process is implemented in the `ast/mod.rs` file.

```rust
// filepath: ast/mod.rs

pub struct Ast {
    pub process_blocks: HashMap<String, (Node<ArgsList>, Node<Block>)
       >,
    pub condition_blocks: HashMap<ConditionKeyword, Node<
       ConditionBlock>>,
    pub global_block: Option<Node<Block>>,
    pub function_blocks: HashMap<String, (Node<ArgsList>, DataType,
       Node<Block>)>,
}

impl Ast {
    pub fn new() -> Self {
        Self {
            process_blocks: HashMap::new(),
            condition_blocks: HashMap::new(),
            global_block: None,
            function_blocks: HashMap::new(),
        }
    }

    pub fn build(pairs: Pairs<Rule>) -> AlthreadResult<Self> {
        let mut ast = Self::new();
        for pair in pairs {
            match pair.as_rule() {
                // Other blocks are removed for brevity
                Rule::function_block => {
                    let mut pairs = pair.into_inner();

                    let function_identifier = pairs.next().unwrap().
                       as_str().to_string();
                    let args_list: Node<token::args_list::ArgsList> =
                        Node::build(pairs.next().unwrap())?;
                    pairs.next(); // skip the "->" token
```

```
                    let return_datatype = DataType::from_str(pairs.next().unwrap().as_str());
                    let function_block: Node<Block> = Node::build(pairs.next().unwrap())?;

                    // Check if the function is already defined
                    if ast.function_blocks.contains_key(&function_identifier) {
                        return Err(AlthreadError::new(
                            ErrorType::FunctionAlreadyDefined,
                            Some(function_block.pos),
                            format!("Function '{}' is already defined", function_identifier),
                        ));
                    }

                    ast.function_blocks.insert(
                        function_identifier,
                        (args_list, return_datatype, function_block),
                    );
                }
                _ => (), // Handle other rules as needed
            }
        }
        Ok(ast)
    }
}

impl AstDisplay for Ast {
    fn ast_fmt(&self, f: &mut Formatter, prefix: &Prefix) -> fmt::Result {
        // other blocks are removed for brevity
        for (function_name, (_args, return_type, function_node)) in &self.function_blocks {
            writeln!(f, "{}{} -> {}", prefix, function_name, return_type)?;
            function_node.ast_fmt(f, &prefix.add_branch())?;
            writeln!(f, "")?;
        }
        Ok(())
    }
}
```

Figure B.2: Althread's full grammar updated to support user-defined functions

Appendix C

Return and Function Call

The full implementation of the return statement is provided in the file `ast/statement/fn_return.rs` which can be viewed in Figure C.1. This code handles the compilation and execution of return statements within user-defined functions.

```rust
// filepath: ast/statement/fn_return.rs

#[derive(Debug, Clone)]
pub struct FnReturn {
    pub value: Option<Node<Expression>>,
    pub pos: Pos,
}

impl NodeBuilder for FnReturn {
    fn build(mut pairs: Pairs<Rule>) -> AlthreadResult<Self> {
        // return statement doesn't necessarily have a value
        let value = if let Some(pair) = pairs.next() {
            Some(Expression::build_top_level(pair)?)
        } else {
            None
        };

        // the caller takes care of setting the proper position
        Ok(Self { value, pos: Pos::default() })
    }
}

impl InstructionBuilder for FnReturn {
    fn compile(&self, state: &mut CompilerState) -> AlthreadResult<
        InstructionBuilderOk> {
        if !state.in_function {
            return Err(AlthreadError::new(
                ErrorType::ReturnOutsideFunction,
                Some(self.pos),
                "Return statement outside function".to_string(),
            ));
        }

        let mut builder = InstructionBuilderOk::new();
        let mut has_value: bool = false;

        if let Some(ref value_node) = self.value {
            builder.extend(value_node.compile(state)?);
            has_value = true;
        }

        let ret_instr = Instruction {
            control: InstructionType::Return {
                has_value
            },
            pos: Some(self.pos),
```

```
        };


        builder.return_indexes.push(builder.instructions.len());

        builder.instructions.push(ret_instr);

        Ok(builder)
    }
}

impl AstDisplay for FnReturn {
    fn ast_fmt(&self, f: &mut fmt::Formatter, prefix: &Prefix) -> fmt
        ::Result {
        writeln!(f, "{prefix}return")?;
        let prefix = prefix.add_branch();

        if let Some(ref value_node) = self.value {
            let prefix_val = prefix.switch();
            writeln!(f, "{}value:", &prefix_val)?;
            value_node.ast_fmt(f, &prefix_val.add_leaf())?;
        } else {
             writeln!(f, "{}(no value)", prefix.switch())?;
        }

        Ok(())
    }
}
```

Figure C.1: Althread's new return statement implementation

The full implementation of the extended function call statement is provided in the file `ast/statement/fn_call.rs` which can be viewed in Figure C.2. This code handles the compilation and execution of function calls, including support for user-defined functions.

```
// filepath: ast/statement/fn_call.rs

#[derive(Debug, Clone, PartialEq)]
pub struct FnCall {
    pub fn_name: Vec<Node<Identifier>>,
    pub values: Box<Node<Expression>>,
}

impl FnCall {
    pub fn add_dependencies(&self, dependencies: &mut WaitDependency)
         {
        for ident in &self.fn_name {
            dependencies.variables.insert(ident.value.value.clone());
        }

        self.values.value.add_dependencies(dependencies);
    }

    pub fn get_vars(&self, vars: &mut HashSet<String>) {
        for ident in &self.fn_name {
```

```
            vars.insert(ident.value.value.clone());
        }

        self.values.value.get_vars(vars);
    }
}

impl NodeBuilder for FnCall {
    fn build(mut pairs: Pairs<Rule>) -> AlthreadResult<Self> {
        let mut object_identifier = pairs.next().unwrap();

        let mut fn_name = Vec::new();

        loop {
            let n: Node<Identifier> = Node::build(object_identifier.
                clone())?;
            fn_name.push(n);

            let mut pairs = object_identifier.into_inner();
            pairs.next().unwrap();
            if let Some(p) = pairs.next() {
                object_identifier = p;
            } else {
                break;
            }
        }

        let values = Box::new(Expression::build_top_level(pairs.next
            ().unwrap())?);

        Ok(Self { fn_name, values })
    }
}

impl InstructionBuilder for Node<FnCall> {
    fn compile(&self, state: &mut CompilerState) -> AlthreadResult<
        InstructionBuilderOk> {

        let mut builder = InstructionBuilderOk::new();
        state.current_stack_depth += 1;

        builder.extend(self.value.values.compile(state)?);

        // normally it's always a tuple so it's always 1 argument
        // Tuple([]) when nothing is passed as argument
        let args_on_stack_var =
            state.program_stack
            .last()
            .cloned()
            .expect("Stack should not be empty");

        // get the function's basename (the last identifier in the
            fn_name)
        let basename = &self.value.fn_name[0].value.value;

        if self.value.fn_name.len() == 1 {

```

```
            if let Some(func_def) = state.user_functions.get(basename).cloned() {

                let expected_args = &func_def.arguments;
                let expected_arg_count = expected_args.len();

                // get the list of arguments (datatypes) from the tuple arg_list
                let provided_arg_types = args_on_stack_var.datatype.tuple_unwrap();

                // check if the number of arguments is correct
                if expected_arg_count != provided_arg_types.len() {

                    state.unstack_current_depth();

                    return Err(AlthreadError::new(
                        ErrorType::FunctionArgumentCountError,
                        Some(self.pos),
                        format!(
                            "Function '{}' expects {} arguments, but {} were provided.",
                            basename,
                            expected_arg_count,
                            provided_arg_types.len()
                        ),
                    ));
                }

                // check if the types of the arguments are correct
                for (i, ((_arg_name, expected_type), provided_type)) in expected_args.iter().zip(provided_arg_types.iter()).enumerate() {
                    if expected_type != provided_type {

                        state.unstack_current_depth();

                        return Err(AlthreadError::new(
                            ErrorType::FunctionArgumentTypeMismatch,
                            Some(self.pos),
                            format!(
                                "Function '{}' expects argument {} ('{}') to be of type {}, but got {}.",
                                basename,
                                i + 1,
                                expected_args[i].0.value, // argument name
                                expected_type,
                                provided_type
                            ),
                        ));
                    }
                }

                let unstack_len = state.unstack_current_depth();

```

```
                builder.instructions.push(Instruction {
                    control: InstructionType::FnCall {
                        name: basename.to_string(),
                        unstack_len,
                        variable_idx: None,
                        arguments: None
                    },
                    pos: Some(self.pos),
                });


                state.program_stack.push(Variable {
                    mutable: true,
                    name: "".to_string(),
                    datatype: func_def.return_type.clone(),
                    depth: state.current_stack_depth,
                    declare_pos: Some(self.pos),
                });

            } else if basename == "print" {

                let unstack_len = state.unstack_current_depth();

                builder.instructions.push(Instruction {
                    control: InstructionType::FnCall {
                        name: basename.to_string(),
                        unstack_len,
                        variable_idx: None,
                        arguments: None,
                    },
                    pos: Some(self.pos),
                });

                state.program_stack.push(Variable {
                    mutable: true,
                    name: "".to_string(),
                    datatype: DataType::Void,
                    depth: state.current_stack_depth,
                    declare_pos: Some(self.pos),
                });

            } else {

                return Err(AlthreadError::new(
                    ErrorType::UndefinedFunction,
                    Some(self.pos),
                    format!("undefined function {}", basename),
                ));
            }

        } else {
            // this is a method call

            //get the type of the variable in the stack with this name
            let var_id = state
                .program_stack
```

```rust
                .iter()
                .rev()
                .position(|var| var.name.eq(basename))
                .ok_or(AlthreadError::new(
                    ErrorType::VariableError,
                    Some(self.pos),
                    format!("Variable '{}' not found", basename),
                ))?;
            let var = &state.program_stack[state.program_stack.len() - var_id - 1];

            let interfaces = state.stdlib.interfaces(&var.datatype);

            // retreive the name of the function
            let fn_name = self.value.fn_name.last().unwrap().value.value.clone();

            let fn_idx = interfaces.iter().position(|i| i.name == fn_name);
            if fn_idx.is_none() {
                return Err(AlthreadError::new(
                    ErrorType::UndefinedFunction,
                    Some(self.pos),
                    format!("undefined function {}", fn_name),
                ));
            }
            let fn_idx = fn_idx.unwrap();
            let fn_info = &interfaces[fn_idx];
            let ret_type = fn_info.ret.clone();

            let unstack_len = state.unstack_current_depth();

            state.program_stack.push(Variable {
                mutable: true,
                name: "".to_string(),
                datatype: ret_type,
                depth: state.current_stack_depth,
                declare_pos: None,
            });

            builder.instructions.push(Instruction {
                control: InstructionType::FnCall {
                    name: fn_name,
                    unstack_len: unstack_len,
                    variable_idx: Some(var_id),
                    arguments: None, // use the top of the stack
                },
                pos: Some(self.pos),
            });
        }

        Ok(builder)
    }
}

impl AstDisplay for FnCall {
    fn ast_fmt(&self, f: &mut fmt::Formatter, prefix: &Prefix) -> fmt
```

```
        ::Result {
         let names: Vec<String> = self.fn_name
             .iter()
             .map(|n| n.value.value.clone())
             .collect();
         let fn_name = names.join(".");
         writeln!(f, "{}{}", prefix, fn_name)?;
         self.values.ast_fmt(f, &prefix.add_leaf())?;

         Ok(())
     }
}
```

Figure C.2: Althread's extended function call statement implementation

Appendix D

Updated VM Code

The main modifications of the virtual machine (VM) for Althread to support user-defined functions are provided below. This code is implemented in the `vm/running_program.rs` file.

```
// filepath: vm/running_program.rs

#[derive(Debug, Clone)]
struct StackFrame<'a> {
return_ip: usize, // the instruction pointer to return to
caller_fp: usize,
caller_code: &'a [Instruction], // the code of the caller
expected_return_type: DataType  // the expected return type of the function
}

#[derive(Debug, Clone)]
pub struct RunningProgramState<'a> {
pub name: String,

memory: Memory,
code: &'a ProgramCode,  // full code
current_code: &'a [Instruction], // current executing code
instruction_pointer: usize,
pub id: usize,
pub stdlib: Rc<Stdlib>,

pub user_functions: &'a HashMap<String, FunctionDefinition>,
call_stack: Vec<StackFrame<'a>>, // the call stack
frame_pointer: usize,
}

impl PartialEq for RunningProgramState<'_> {
fn eq(&self, other: &Self) -> bool {
    self.id == other.id
        && self.memory == other.memory
        && self.name == other.name
        && self.instruction_pointer == other.instruction_pointer
        && self.frame_pointer == other.frame_pointer
        && self.call_stack.len() == other.call_stack.len()
}
}

impl Hash for RunningProgramState<'_> {
fn hash<H: Hasher>(&self, state: &mut H) {
    self.id.hash(state);
    self.memory.hash(state);
    self.instruction_pointer.hash(state);
}
}

impl<'a> RunningProgramState<'a> {
```

```rust
pub fn new(
    id: usize,
    name: String,
    code: &'a ProgramCode,
    user_functions: &'a HashMap<String, FunctionDefinition>,
    args: Literal,
    stdlib: Rc<Stdlib>,
) -> Self {
    let arg_len = if let Literal::Tuple(v) = &args {
        v.len()
    } else {
        panic!("args should be a tuple")
    };

    let memory = if arg_len > 0 { vec![args] } else { Vec::new() };

    Self {
        id,
        name,
        memory,
        code,
        current_code: &code.instructions,
        instruction_pointer: 0,
        stdlib,
        user_functions,
        call_stack: Vec::new(),
        frame_pointer: 0,
    }
}

pub fn current_state(&self) -> (&Memory, usize) {
    (&self.memory, self.instruction_pointer)
}

pub fn current_instruction(&self) -> AlthreadResult<&Instruction> {
    self.current_code
        .get(self.instruction_pointer)
        .ok_or(AlthreadError::new(
            ErrorType::InstructionNotAllowed,
            None,
            format!(
            "the current instruction pointer points to no instruction
                (pointer:{}, program:{})",
            self.instruction_pointer, self.name
        ),
        ))
}

pub fn has_terminated(&self) -> bool {
    if let Some(inst) = self.current_instruction().ok() {
        inst.is_end()
    } else {
        true
    }
}

pub fn next_global(
```

```
    &mut self,
    globals: &mut GlobalMemory,
    channels: &mut Channels,
    next_pid: &mut usize,
) -> AlthreadResult<(GlobalActions, Vec<Instruction>)> {
    let mut instructions = Vec::new();
    let mut actions = Vec::new();
    let mut wait = false;
    let mut end = false;
    loop {
        let (at_actions, at_instructions) = self.next_atomic(globals,
             channels, next_pid)?;

        actions.extend(at_actions.actions);
        instructions.extend(at_instructions);

        if at_actions.wait {
            wait = true;
            break;
        }
        if at_actions.end {
            end = true;
            break;
        }
        if self.is_next_instruction_global() {
            break;
        }
    }
    Ok((GlobalActions { actions, wait, end }, instructions))
}

pub fn is_next_instruction_global(&mut self) -> bool {
    self.current_instruction()
        .map_or(true, |inst| !inst.control.is_local())
}

pub fn next_atomic(
    &mut self,
    globals: &mut GlobalMemory,
    channels: &mut Channels,
    next_pid: &mut usize,
) -> AlthreadResult<(GlobalActions, Vec<Instruction>)> {
    let mut instructions = Vec::new();

    let mut result = GlobalActions {
        actions: Vec::new(),
        wait: false,
        end: false,
    };
    // if the next instruction is not the start of an atomic block,
       we execute the next instruction
    if !self.current_instruction()?.is_atomic_start() {
        instructions.push(self.current_instruction()?.clone());
        let action = self.next(globals, channels, next_pid)?;
        if let Some(action) = action {
            if action == GlobalAction::Wait {
                result.wait = true;
```

```
            } else if action == GlobalAction::EndProgram {
                result.end = true;
            } else {
                result.actions.push(action);
            }
        }
        return Ok((result, instructions));
    }
    // else we execute all the instructions until the end of the atomic block
    loop {
        instructions.push(self.current_instruction()?.clone());
        let action = self.next(globals, channels, next_pid)?;
        if let Some(action) = action {
            if action == GlobalAction::Wait {
                result.wait = true;
                break;
            } else {
                result.actions.push(action);
            }
        }
        if self.current_instruction()?.is_atomic_end() {
            break;
        }
    }
    Ok((result, instructions))
}

fn next(
    &mut self,
    globals: &mut GlobalMemory,
    channels: &mut Channels,
    next_pid: &mut usize,
) -> AlthreadResult<Option<GlobalAction>> {

    let cur_inst = self.current_instruction()?.clone();

    let mut action = None;

    let pos_inc = match &cur_inst.control {
        InstructionType::Empty => 1,
        InstructionType::AtomicStart => 1,
        InstructionType::AtomicEnd => 1,
        InstructionType::Break {
            unstack_len, jump, ..
        } => {
            for _ in 0..*unstack_len {
                self.memory.pop();
            }
            *jump
        }
        InstructionType::JumpIf {
            jump_false,
            unstack_len,
        } => {
            let cond = self.memory.last().unwrap().is_true();
            for _ in 0..*unstack_len {
```

```
                self.memory.pop();
            }
            if cond {
                1
            } else {
                *jump_false
            }
        }
        InstructionType::Jump(jump) => *jump,
        InstructionType::Expression(exp) => {
            let lit = exp.eval(&mut self.memory).map_err(|msg| {
                AlthreadError::new(ErrorType::ExpressionError, cur_inst.pos, msg)
            })?;
            self.memory.push(lit);
            1
        }
        InstructionType::GlobalReads { variables, .. } => {
            for var_name in variables.iter() {
                self.memory.push(
                    globals
                        .get(var_name)
                        .expect(format!("global variable '{}' not found", var_name).as_str())
                        .clone(),
                );
            }
            1
        }
        InstructionType::GlobalAssignment {
            identifier,
            operator,
            unstack_len,
        } => {
            let lit = self
                .memory
                .last()
                .expect("Panic: stack is empty, cannot perform assignment")
                .clone();
            for _ in 0..*unstack_len {
                self.memory.pop();
            }

            let lit = operator
                .apply(
                    &globals
                        .get(identifier)
                        .expect(format!("global variable '{}' not found", identifier).as_str()),
                    &lit,
                )
                .map_err(str_to_expr_error(cur_inst.pos))?;

            globals.insert(identifier.clone(), lit);
            action = Some(GlobalAction::Write(identifier.clone()));
            1
```

```
        }
        InstructionType::LocalAssignment {
            index,
            unstack_len,
            operator,
        } => {
            let lit = self
                .memory
                .last()
                .expect("Panic: stack is empty, cannot perform
                    assignment")
                .clone();
            for _ in 0..*unstack_len {
                self.memory.pop();
            }

            let len = self.memory.len();

            self.memory[len - 1 - index] = operator
                .apply(&self.memory[len - 1 - *index], &lit)
                .map_err(str_to_expr_error(cur_inst.pos))?;
            1
        }
        InstructionType::Unstack { unstack_len } => {
            for _ in 0..*unstack_len {
                self.memory.pop();
            }
            1
        }
        InstructionType::Declaration { unstack_len } => {
            let lit = self
                .memory
                .last()
                .expect("Panic: stack is empty, cannot perform
                    declaration with value")
                .clone();
            for _ in 0..*unstack_len {
                self.memory.pop();
            }
            self.memory.push(lit);
            1
        }
        InstructionType::RunCall { name, unstack_len } => {
            let args = self
                .memory
                .last()
                .expect("Panic: stack is empty, cannot run call")
                .clone();
            for _ in 0..*unstack_len {
                self.memory.pop();
            }
            self.memory.push(Literal::Process(name.clone(), *next_pid
                ));
            action = Some(GlobalAction::StartProgram(name.clone(), *
                next_pid, args));
            *next_pid += 1;
            1
```

```
        }
        InstructionType::EndProgram => {
            if self.call_stack.is_empty() {
                action = Some(GlobalAction::EndProgram);
                0
            } else {
                let return_value = Literal::Null;
                let frame = self.call_stack.pop().unwrap();
                self.memory.truncate(self.frame_pointer);
                self.frame_pointer = frame.caller_fp;
                self.instruction_pointer = frame.return_ip;
                self.current_code = &self.code.instructions;
                self.memory.push(return_value);
                0
            }
        }
        InstructionType::Return {has_value}  => {

            let return_value = if *has_value {
                self.memory.pop().expect("Stack empty, expected return value")
            } else {
                Literal::Null
            };


            let frame = self.call_stack.pop().expect("Panic: stack is empty, cannot perform return");

            if return_value.get_datatype() != frame.expected_return_type {
                return Err(AlthreadError::new(
                    ErrorType::FunctionReturnTypeMismatch,
                    cur_inst.pos,
                    format!(
                        "expected {:?}, got {:?}",
                        frame.expected_return_type,
                        return_value.get_datatype()
                    ),
                ));
            }

            self.memory.truncate(self.frame_pointer);

            self.frame_pointer = frame.caller_fp;
            self.instruction_pointer = frame.return_ip;
            self.current_code = frame.caller_code;

            self.memory.push(return_value);

            0
        }
        InstructionType::FnCall {
            variable_idx,
            name,
            arguments,
            unstack_len,
```

```
        } => {
            if let Some(v_idx) = variable_idx {
                let v_idx = self.memory.len() - 1 - v_idx;
                let mut lit = self
                    .memory
                    .get(v_idx)
                    .expect("Panic: stack is empty, cannot perform function call")
                    .clone();

                let interfaces = self.stdlib.get_interfaces(&lit.get_datatype()).ok_or(
                    AlthreadError::new(
                        ErrorType::UndefinedFunction,
                        cur_inst.pos,
                        format!("Type {:?} has no interface available", lit.get_datatype()),
                    ),
                )?;

                let fn_idx = interfaces.iter().position(|i| i.name == *name);
                if fn_idx.is_none() {
                    return Err(AlthreadError::new(
                        ErrorType::UndefinedFunction,
                        cur_inst.pos,
                        format!("undefined function {}", name),
                    ));
                }
                let fn_idx = fn_idx.unwrap();
                let interface = interfaces.get(fn_idx).unwrap();
                let mut args = match &arguments {
                    None => self.memory.last().unwrap().clone(),
                    Some(v) => {
                        let mut args = Vec::new();
                        for i in 0..v.len() {
                            let idx = self.memory.len() - 1 - v[i];
                            args.push(self.memory.get(idx).unwrap().clone());
                        }
                        Literal::Tuple(args)
                    }
                };
                let ret = interface.f.as_ref()(&mut lit, &mut args);

                //update the memory with object literal
                self.memory[v_idx] = lit;

                for _ in 0..*unstack_len {
                    self.memory.pop();
                }

                self.memory.push(ret);
                1
            } else {
                // currently, only the print function is implemented
                if name == "print" {
```

```
    let lit = self
        .memory
        .last()
        .expect("Panic: stack is empty, cannot perform function call")
        .clone();

    for _ in 0..*unstack_len {
        self.memory.pop();
    }

    let str_val = lit.into_tuple().unwrap_or_default()
        .iter()
        .map(|lit| lit.to_string())
        .collect::<Vec<_>>()
        .join(",");
    println!("{}", str_val);
    action = Some(GlobalAction::Print(str_val));
    self.memory.push(Literal::Null);
    1
} else {
if let Some(func_def) = self.user_functions.get(name) {

    let args_tuple_lit = self.memory.pop().unwrap();
    let arg_values = match args_tuple_lit {
        Literal::Tuple(v) => v,
        _ => {
            return Err(AlthreadError::new(
                ErrorType::RuntimeError,
                cur_inst.pos,
                format!("function {} expects a tuple as argument", name),
            ));
        }
    };

    self.call_stack.push(StackFrame {
        return_ip: self.instruction_pointer + 1,
        caller_fp: self.frame_pointer,
        caller_code: self.current_code,
        expected_return_type: func_def.return_type.clone(),
    });

    self.frame_pointer = self.memory.len();

    for arg in arg_values {
        self.memory.push(arg);
    }

    self.current_code = &func_def.body;
    self.instruction_pointer = 0;

    0
} else {
```

```
                    return Err(AlthreadError::new(
                        ErrorType::UndefinedFunction,
                        cur_inst.pos,
                        format!("undefined function {}", name),
                    ));
                }
            }
        }
    }
    InstructionType::WaitStart { .. } => 1,
    InstructionType::Wait {
        unstack_len, jump, ..
    } => {
        let cond = self.memory.last().unwrap().is_true();
        for _ in 0..*unstack_len {
            self.memory.pop();
        }
        if cond {
            1
        } else {
            action = Some(GlobalAction::Wait);
            *jump
        }
    }
    InstructionType::Destruct => {
        // The values are in a tuple on the top of the stack
        let tuple = self
            .memory
            .pop()
            .expect("Panic: stack is empty, cannot destruct")
            .into_tuple()
            .expect("Panic: cannot convert to tuple");
        for val in tuple.into_iter() {
            self.memory.push(val);
        }
        1
    }
    InstructionType::Push(literal) => {
        self.memory.push(literal.clone());
        1
    }
    InstructionType::Send {
        channel_name,
        unstack_len,
    } => {
        let value = self
            .memory
            .last()
            .expect("Panic: stack is empty, cannot send")
            .clone();

        for _ in 0..*unstack_len {
            self.memory.pop();
        }

        let receiver = channels.send(self.id, channel_name.clone
            (), value);
```

```
                action = Some(GlobalAction::Send(channel_name.clone(),
                    receiver));
                1
            }
            InstructionType::ChannelPeek(channel_name) => {
                let values = channels.peek(self.id, channel_name.clone())
                    ;
                match values {
                    Some(value) => {
                        self.memory.push(value.clone());
                        self.memory.push(Literal::Bool(true));
                    }
                    None => {
                        self.memory.push(Literal::Bool(false));
                    }
                }
                1
            }
            InstructionType::ChannelPop(channel_name) => {
                let _ = channels.pop(self.id, channel_name.clone());
                1
            }
            InstructionType::Connect {
                sender_pid,
                sender_channel,
                receiver_pid,
                receiver_channel,
            } => {
                let sender_pid = match *sender_pid {
                    None => self.id,
                    Some(idx) => self
                        .memory
                        .get(self.memory.len() - 1 - idx)
                        .expect("Panic: stack is empty, cannot connect")
                        .clone()
                        .to_pid()
                        .expect("Panic: cannot convert to pid"),
                };
                let receiver_pid = match receiver_pid {
                    None => self.id,
                    Some(idx) => self
                        .memory
                        .get(self.memory.len() - 1 - idx)
                        .expect("Panic: stack is empty, cannot connect")
                        .clone()
                        .to_pid()
                        .expect("Panic: cannot convert to pid"),
                };

                let is_data_waiting = channels
                    .connect(
                        sender_pid,
                        sender_channel.clone(),
                        receiver_pid,
                        receiver_channel.clone(),
                    )
```

```
                .map_err(|msg| {
                    AlthreadError::new(ErrorType::RuntimeError, cur_inst.pos, msg)
                })?;
            // A connection has the same effect as a send globally, if some data was waiting to be sent
            if is_data_waiting {
                action = Some(GlobalAction::Send(
                    sender_channel.clone(),
                    Some(ReceiverInfo {
                        program_id: receiver_pid,
                        channel_name: receiver_channel.clone(),
                    }),
                ));
            }
            1
        }
        _ => panic!("Instruction '{:?}' not implemented", cur_inst.control),
    };
    let new_pos = (self.instruction_pointer as i64) + pos_inc;
    if new_pos < 0 {
        return Err(AlthreadError::new(
            ErrorType::RuntimeError,
            None,
            "instruction pointer is becomming negative".to_string(),
        ));
    }
    self.instruction_pointer = new_pos as usize;
    Ok(action)
}
}
```

Figure D.1: Althread's VM implementation updated to support user-defined functions.

Appendix E

Tests

```
fn fibonacci_recursive(n: int, a: int, b: int) -> int {
  if n == 0 {
    return a;
  } else {
    return fibonacci_recursive(n - 1, b, a + b);
  }
}

fn fibonacci_iterative(n: int, a: int, b: int) -> int {
  for i in 1..n {
    let c = a + b;
    a = b;
    b = c;
  }
  return b;
}

main {
    let n = 10;
    let res = fibonacci_recursive(n, 0, 1);
    print("Fibonacci recursive of " + n + ": " + res);

    let res = fibonacci_iterative(n, 0, 1);
    print("Fibonacci iterative of " + n + ": " + res);
}

// Outputs:
// Fibonacci recursive of 10: 55
// Fibonacci iterative of 10: 55
```

Figure E.1: Testing user-defined functions by implementing both recursive and iterative versions of the Fibonacci sequence.

```
shared {
    let A = 1;
    let B = 0;
    let Start = false;
    let WorkersFinished = 0;   // Counts finished workers
}

fn process_message(value: int, flag: bool) -> void {
    print("Processing message: value=" + value + ", flag=" + flag);
    atomic {
        if flag {
            A = value;
        } else {
            B = value;
        }
```

```
        WorkersFinished += 1;
    }
}

fn verify_state() -> bool {
    return (A == 125 && B == 125);
}

program Worker() {
    wait Start;
    wait receive in (x, y) => {
        process_message(x, y);
    };
}

main {
    let worker1 = run Worker();
    let worker2 = run Worker();

    channel self.out (int, bool)> worker1.in;
    channel self.out2 (int, bool)> worker2.in;

    atomic { Start = true; }

    send out(125, true);
    send out2(125, false);

    // Waits for both workers to finish processing
    wait WorkersFinished == 2;

    if verify_state() {
        print("Channel test successful!");
    } else {
        print("Channel test failed!");
    }
}

// Output:
// Processing message: value=125, flag=true
// Processing message: value=125, flag=false
// Channel test successful!
// or
// Processing message: value=125, flag=false
// Processing message: value=125, flag=true
// Channel test successful!
```

Figure E.2: This test demonstrates the use of atomic blocks, conditional statements, and access to shared variables in user-defined functions. Two worker processes are spawned, each waiting to receive a message. Upon receiving a message, each worker updates shared variables within an atomic block to ensure that updates are performed without interference from other processes.